EVALUATING THE EFFECT OF ISCSI PROTOCOL PARAMETERS ON PERFORMANCE

Yamini Shastry, Steve Klotz
Medusa Labs
600 Center Ridge Drive, Suite 600
Austin, Texas 78753, USA
email: {yamini,steve}@medusalabs.com

Robert D. Russell
InterOperability Lab, Univ. of New Hampshire
121 Technology Drive, Suite 2
Durham, NH 03824, USA
email: rdr@iol.unh.edu

ABSTRACT
iSCSI is a new IETF standard protocol that makes it possible for a SCSI initiator (client) on one machine to exchange SCSI commands and data with a SCSI target (server) on another machine connected via a TCP/IP network. When an initiator establishes a connection to a target, a large number of standard parameters can be negotiated in order to customize many aspects of that connection. Both the initiator and target implementations can configure other parameters outside the iSCSI standard that also customize their interactions with the SCSI environment.

This paper reports the major results from an extensive study of the effect of these parameters on iSCSI performance. It also develops a linear relationship between a number of them. We suggest settings that offer the best performance in the situations tested, with the belief that these settings offer the best available general guidance for configuring iSCSI until results from testing in more complex situations becomes available.

KEY WORDS
protocols, performance evaluation, storage area networks, iSCSI.

1 Introduction

The iSCSI protocol is a mapping of the SCSI remote invocation procedure model [1] onto the TCP/IP protocol suite. The iSCSI specifications have recently been approved as an *Internet Engineering Task Force* (IETF) standard [2]. The strength of iSCSI stems from the fact that it builds on well-established technologies like SCSI, TCP/IP and Ethernet. Native SCSI, as used in direct attached storage, is both a protocol and a physical transport. iSCSI is a session-based protocol in which the transport is provided by TCP/IP. iSCSI uses a client-server architecture in which the client is called the iSCSI initiator and the server is called the iSCSI target. The design of the iSCSI protocol is explained in [3]. An important part of that design is the specification of a large number of operational parameters which can be negotiated on a session by session basis.

A number of papers have discussed various aspects of the performance of the iSCSI protocol [4][5][6][7]. However, of these only [7] seems to have looked at the effects which tuning the various settings of the standard iSCSI parameters may have on performance. The others do not even state what their iSCSI parameter settings were during their tests.

This paper presents the major results from an in-depth study of the iSCSI parameters defined in the standard [2]. Section 2 introduces and categorizes these parameters, as well as some other parameters which are typically found in iSCSI implementations. Section 3 discusses the performance metrics used in our study, and section 4 describes our test setup. Because they involve different parameters, our results for read operations are found in section 5, while those for write operations are found in section 6. Section 7 summarizes the conclusions and suggests some future work.

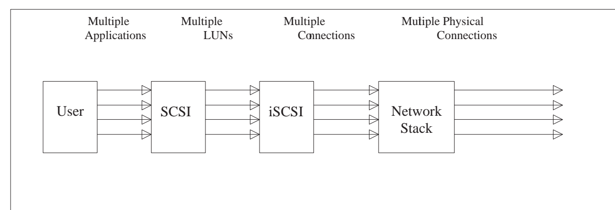

Figure 1. Multiplexing Single/Multiple connections/LUs

2 Parameters

Each iSCSI connection starts with a *login phase* during which the initiator and target may negotiate a large number of standard parameters that can effect performance. Both initiator and target may also independently define a number of implementation-dependent parameters that effect performance. This second set of parameters cannot be negotiated during login, unlike the standard parameters.

Once the login phase is successfully completed, iSCSI enters the *full feature phase*, during which iSCSI commands and data are exchanged over the established iSCSI connections. The unit of transmission in iSCSI is the *Protocol Data Unit* (PDU). Data transfer is performed during *read* commands, in which an initiator receives data from a target, and *write* commands, in which data is sent from initiator to target.

2.1 Parameters effecting all commands

- **Header and Data Digests**: A 32-bit *Cyclic Redundancy Check (CRC)* may be added to the end of iSCSI headers and/or iSCSI data payloads in order to catch errors that are undetected by the weak checksum of TCP. Although the CRC calculation was performed in software for our tests, it had no effect on throughput for small PDU sizes (8192 bytes or less), but moderately decreased throughput for larger PDUs.

- **DataPDUInOrder and DataSequenceInOrder**: These parameters may be negotiated in order to restrict the order of data pdus within sequences, and the order of the sequences themselves. We found that these parameters had no measurable effect on performance when used with our test targets. A more sophisticated target using a storage device that could control out-of-order operations in order to reduce its processing time would need to be found and tested.

- **MaxConnections**: This gives the maximum number of TCP connections allowed in a single session. If this is negotiated to a value greater than 1, then a further consideration not defined in the standard is how these connections are mapped onto physical interfaces. With only one physical interface, we found that multiple connections per session had no effect on throughput.

- **Number of Logical Units**: The number of Logical Units (LUs) defined by a target can effect performance, because a single connection may be used to carry commands and data for multiple LUs. When there are multiple connections in the session, there can be different combinations for setting them up, as shown in Figure 1. We found that multiple LUs had no measurable effect on performance.

- **Initiator connection scheduling**: An important consideration in implementing an initiator that allows multiple connections per session is the design of algorithms which decide how and when to map commands onto connections. To date, only the Round Robin and LU-assignment-to-connection algorithms have been implemented in the UNH reference initiator, and neither algorithm showed any performance benefit over the other in our tests.

- **Number of outstanding SCSI commands**: Most SCSI implementations permit an initiator to have more than one SCSI command in progress simultaneously. The exact number is a non-negotiable property of the iSCSI initiator. We found that it has a significant impact on throughput.

- **Number of sectors per command**: Most SCSI disks define a sector size of 512 bytes, and require I/O operations to be in multiples of a sector. The iSCSI initiator is usually required to declare a limit on the number of sectors in a single SCSI I/O operation. Increasing the value of this parameter results in an improvement in throughput.

2.2 Parameters effecting read commands

Three parameters that potentially could affect the performance of an iSCSI read are *MaxBurstLength* (MBL), *phase collapse*, and *MaxRecvDataSegmentLength* (MRDSL) of the initiator.

On receiving a read command, the target sends the requested data to the initiator. This data is split into sequences. Each sequence consists of one or more data-in pdus. The total amount of read data sent in one sequence must not exceed the negotiated value of the *MaxBurstLength*. The total amount of data sent in all sequences must equal the amount of data requested in the read command (the read command's *ExpectedDataTransferLength*). The *DataSegmentLength* of a data-in pdu must not exceed *MRDSL* of the initiator, and the total of all the *DataSegmentLength*s of all the pdus in a sequence must not exceed the *MaxBurstLength*.

MaxBurstLength has a significant impact on write performance, as shown in section 7, but no impact on read performance. This requires a bit of explanation. For both reading and writing, *MaxBurstLength* determines the length of a sequence of data-carrying pdus. However, the notion of a sequence during a read is almost meaningless in the iSCSI protocol, because the last data-in pdu at the end of one input sequence can be immediately followed by the first data-in pdu at the beginning of the next, with no extra pdu exchanges in either direction, and no extra processing on either side. By contrast, the end of a sequence of data-out pdus sent by the initiator during writing requires the transmission of an R2T pdu by the target back to the initiator before the next data-out sequence can begin. This extra interaction introduces extra overhead and delay during a write that is completely missing from a read.

The target normally finishes a read command by sending a separate SCSI response pdu containing the command's status. However, when the status is *success*, a non-negotiable option allows the target to use a *phase collapse* in which it sets a bit in the final data-in pdu and omits the SCSI response pdu.

2.3 Parameters effecting write commands

The performance of iSCSI writes depends on the following negotiable iSCSI parameters:

- For unsolicited write data: *FirstBurstLength*, *ImmediateData* (yes/no), *InitialR2T* (yes/no), and *MaxRecvDataSegmentLength* of the target;

- For solicited write data: *MaxBurstLength*, *MaxOutstandingR2T*, and *MaxRecvDataSegmentLength* of the target.

The initiator sends a write command to the target when it has data to be written to the target. The outgoing data follows the write command on the same connection. After the data is written to target, the target sends a response with the status of the write operation.

Data sent in a write command can either be *solicited* or *unsolicited*. Use of unsolicited data is negotiable, and can be sent as immediate data in the write command pdu (when *ImmediateData* is *Yes*), or as separate data-out pdus following the write pdu (when *InitialR2T* is *No*), or both. Solicited data follows any unsolicited data, and is sent by an initiator only after receiving a *ReadyToTransfer* (R2T) pdu from the target. The total amount of solicited and unsolicited data to be written is given by the *ExpectedDataTransferLength (EDTL)* field in the write command pdu.

Both solicited and unsolicited data is written in sequences containing one or more data-out pdus, and the *DataSegmentLength (DSL)* of each data-out pdu must not exceed *MRDSL* of the target. The total of immediate data and all the *DSLs* of all pdus in an unsolicited data sequence is limited by the value of the *FirstBurstLength* parameter. The total amount of all the *DSLs* of all pdus in a solicited data sequence is limited by the value of the *MaxBurstLength* parameter.

When the *MaxOutstandingR2T* parameter is negotiated to a value greater than one, the target can send that number of R2Ts without waiting for any data-out pdus from the initiator, and the initiator can then send that many sequences of data-out pdus without waiting for further input from the target.

3 Performance metrics

Performance metrics considered during our studies were the percent CPU utilization, the command completion time (a measure of latency), the average throughput on an iSCSI connection, and the burst data rates on a command. In this paper, we discuss only results related to average throughput, which is the total number of application-level bytes carried over an iSCSI connection divided by the total elapsed time taken by the application, as expressed in Megabytes per second (MB/sec). Throughout this paper, KB represents $2^{10} = 1,024$ bytes, and MB represents $2^{20} = 1,048,576$ bytes. (Increase all MB/sec numbers in the paper by 5% to get the equivalent value when MB represents $10^6 = 1,000,000$ bytes.)

4 Testing

4.1 Test setup

Our test setup consists of an initiator and target connected point-to-point via an analyzer, as shown in Figure 2. The hardware for the initiator and target systems consisted of 2.66GHz Intel Pentium 4 processors with 512MB of RAM and 32-bit PCI buses. Each processor had a 32-bit wide Intel Pro/1000 82450 gigabit (copper) ethernet adapter. The analyzer was a 1.6GHz AMD Athlon with 256MB of RAM and two Finisar GBE-0901 gigabit ethernet adapters that were internally connected so that no delay was encountered in the analyzer.

Some of the tests were repeated with an Intel NetStructure 470T gigabit ethernet switch inserted between the initiator and target, but no difference in performance was detected.

Although this test setup is very simple, performance measured with it should form a baseline against which performance on more complex network topologies can be compared. Furthermore, our study should serve to guide future performance studies by indicating which of the many iSCSI parameters seem to be the most interesting to look at.

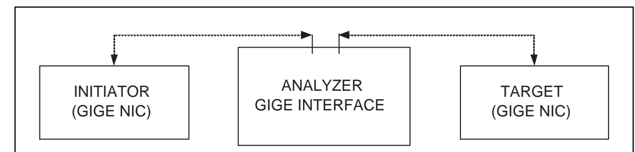

Figure 2. Test Setup

4.2 iSCSI components

The software consisted of the UNH-IOL iSCSI initiator and target-emulator reference implementations [8], the Windows initiator [9], and three other targets, which we refer to as vendor-A target, vendor-B target and vendor-C target because we cannot disclose the product names and details.

Except for tests involving the Windows initiator, the Linux kernel 2.4.18-3 (Redhat 9 distribution) was installed on the initiator. When using the UNH-IOL initiator, data was read and written using raw devices in order to by-pass buffer and file system caching. When using the UNH-IOL target-emulator, which was also implemented on the Linux kernel 2.4.18-3 Redhat 9 distribution, data was kept in RAM (i.e., it was not written to an actual device) so that the results would reflect the performance of the protocol stacks, not that of a storage device.

4.3 Testing methodology

We used a Finisar Xgig iSCSI analyzer in order to measure throughput, data burst rate and command time. Total elapsed time was measured as the difference between the time at which the test program started generating the first byte of data and the time at which the last byte of data was confirmed to have been sent (received). The elapsed time therefore includes all data transfers and all read and write commands and responses at all levels of the protocol stack. For these tests we used custom programs de-

veloped at UNH and Medusa Labs to generate streams of iSCSI read and write data. These were run as user programs on the initiator machine, and in each run the test program continuously read (wrote) 500 requests of one megabyte each as fast as possible through the SCSI subsytem and out over the iSCSI connection to the target device. Each data point plotted in a graph in this paper was calculated as the average of 10 runs with identical parameter settings. We note that in most operating systems, all user-level requests which go through the SCSI subsystem are reorganized into one or more SCSI commands, the size of which bears little relation to the original size of the user-level request, since requests can be broken into smaller pieces or combined into larger ones by the SCSI subsystem. Therefore, we do not vary or plot the user-level request size, since tests results based on the size of requests generated by an application tool are usually meaningless, although they are often reported in the literature when standard data generators are used.

5 iSCSI read performance results

This section shows the effect on iSCSI read throughput of the most important parameters discussed in section 2.

5.1 Effect of MaxRecvDataSegmentLength

To study the effect of *MaxRecvDataSegmentLength* (MRDSL), the *MaxBurstLength* was kept constant at 128KB and the *MRDSL* value was varied from 512 bytes to 128KB. Figure 3 shows a quick 15% increase in throughput as pdu size increases from 512 to 8192 bytes, after which there is only a slight increase in throughput with increased pdu size.

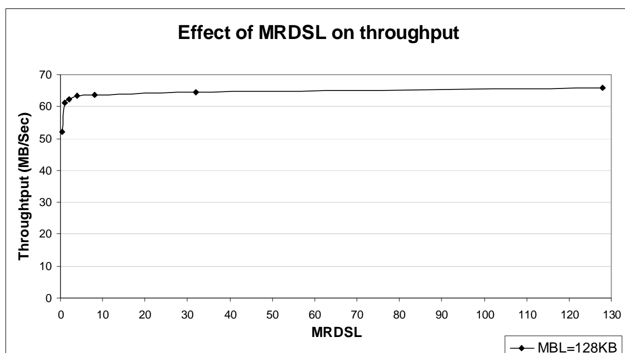

Figure 3. Effect of MRDSL on throughput

The number of data-in pdus generated by a read command is the command size divided by *MRDSL*. Hence, bigger *MRDSL* requires fewer data-in pdus, fewer data-in pdus means fewer iSCSI packets and less overhead processing time. The number of data-ins for each of the points plotted in Figure 3 was verified with counters in the analyzer, and Figure 3 was re-plotted in Figure 4 using the number of data-ins per command instead of the pdu sizes on the x axis. Since this graph shows a nearly linear relation, the following linear equation was developed by regression analysis:

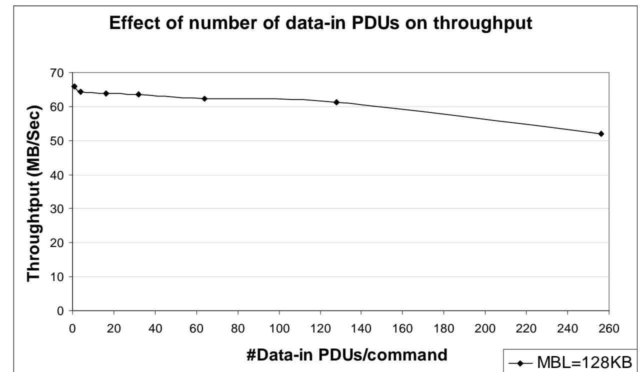

Figure 4. Effect of number of data-in pdus per command

$$t = A * x + B \quad (1)$$

where:

t = Time in milliseconds to transmit 1 MB of data
x = Number of data-in pdus per command
A = 0.014169577
B = 15.24598802

Figure 5 shows the observed values and the values calculated with equation 1 for the time taken to transmit one megabyte of data (i.e., the inverse of throughput) vs the number of data-ins per command. The calculated values show a good linear fit to the actual values. The same comparison was also performed using the Windows initiator and the UNH target, and also showed a good fit.

This test was repeated using the UNH initiator against vendor-A target, vendor-B target and vendor-C target. All the graphs showed the same linear relationship between *MRDSL* and transmission time per MB, but yielded different coefficients for equation 1. This was because the target back-end processing time changed with the target and therefore changed the throughput values.

5.2 Effect of number of sectors per command

Figure 6 shows that throughput increases as the number of sectors per command increases. Increasing the number of sectors which can be read per command means a corresponding increase in the number of bytes of data that can be transfered in a single command, which in turn means fewer commands are required to transfer a fixed amount of data. The number of sectors/command was increased from 64 to 1024, which increases the expected command size from 32KB to 512KB. The burst lengths and pdu sizes were kept equal to the expected command size in order

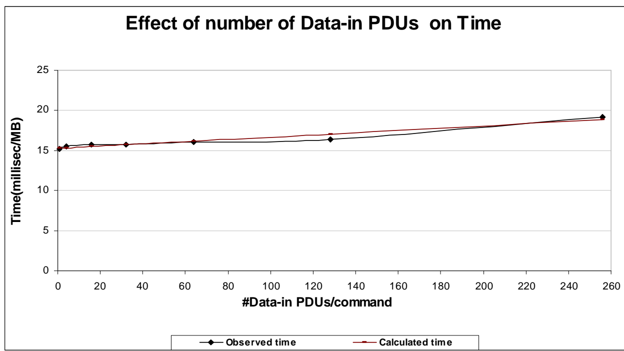

Figure 5. Observed and calculated values for time/MB

to have one response pdu per command pdu. The results show that the throughput does not increase when the number of sectors/command is greater than 1024, because beyond 1024 sectors/command, the SCSI command size does not increase with an increase in sectors/command. This might be a Linux SCSI limitation.

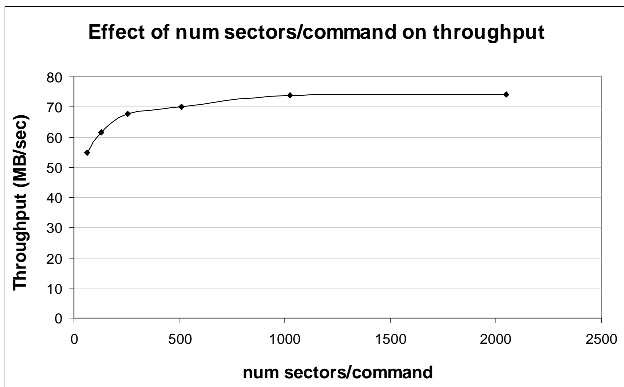

Figure 6. Effect of number of sectors on throughput

6 iSCSI write performance results

This section shows the effect on iSCSI write throughput of the most important parameters discussed in section 2.

6.1 Effect of burst lengths

Figure 7 shows that throughput increases with an increase in burst lengths. Each line in Figure 7 represents a fixed *MRDSL* value. The *FirstBurstLength* was set equal to *MaxBurstLength*. For each value of *MRDSL* (in the range 512 bytes to 128KB), the values of *FirstBurstLength* and *MaxBurstLength* were increased over the same range. This test was repeated for the four combinations of *InitialR2T* and *ImmediateData*, and the results were similar in all cases. Figure 7 shows only the results for *ImmediateData=Yes* and *InitialR2T=No*.

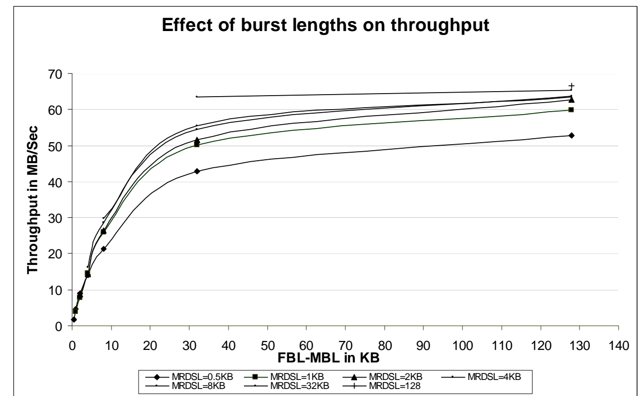

Figure 7. Effect of burst lengths on throughput

Figure 7 shows that for any pdu size, the throughput rapidly increases with an increase in burst length up to 32KB, after which the increase slows down. When repeated with vendor-A and vendor-B targets, this test showed a similar effect.

For any given pdu size, the number of data-outs sent by the initiator in a command is unchanged, but increasing the burst sizes causes more data-out pdus to be sent in a single sequence, which means fewer R2Ts have to be sent by the target to request the sequences from the initiator. Fewer R2Ts means less time spent waiting for and processing R2Ts, and therefore better bandwidth utilization. The number of R2Ts for each point shown the Figure 7 was calculated and the values were verified with counters in the analyzer. Figure 8 is Figure 7 redrawn to show the effect of the number of R2Ts per command on throughput, and dramatically shows the degradation in throughput as this number increases.

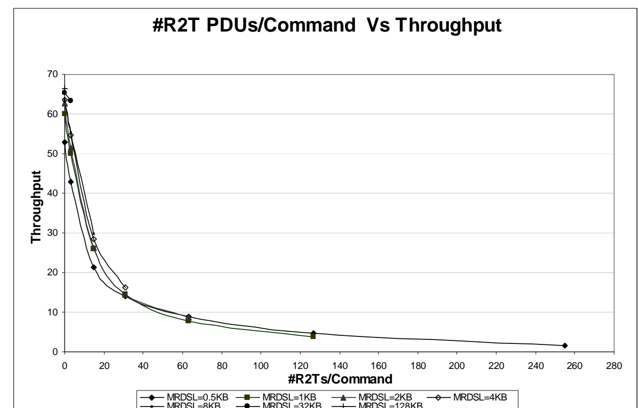

Figure 8. Effect of Number of R2Ts on throughput

6.2 Effect of MaxRecvDataSegmentLength

To study the effect of *MRDSL*, the burst lengths were kept constant at 128KB and the *MRDSL* value was varied from 512 bytes to 128KB. The *ImmediateData* was set to *yes* and *InitialR2T* was set to *no*. This setting was chosen so that all data was sent either immediate or unsolicited, i.e., no R2Ts, and hence the variation in throughput was a pure effect of the pdu size. The results were identical to those for read commands shown in Figures 3 and 4.

From the data collected for all four combinations of *ImmediateData* and *InitialR2T*, it was hypothesized that throughput for write commands depends on the following factors:

- Number of R2Ts per command
- Number of data-out pdus per command
- Number of Immediate data bytes per command
- Number of Unsolicited data bytes per command
- Number of Solicited data bytes per command

A regression analysis was used to develop a linear equation between the inverse of throughput (i.e., the time taken to transmit one megabyte of data), and the above-mentioned factors based on the data from all combinations of *ImmediateData* and *InitialR2T*. Equation 2 shows the result.

$$t = A*x1 + B*x2 + C*x3 + D*X4 + E*x5 + F \quad (2)$$

where:

t = Time in milliseconds to transmit 1 MB of data
x1 = Number of R2Ts per command
x2 = Number of data-outs per command
x3 = Immediate data bytes(MB) per command
x4 = Unsolicited data bytes(MB) per command
x5 = Solicited data bytes(MB) per command
A = 1.82
B = 0.011
C = 115.29
D = 120.79
E = 87.72
F = 0

Figure 9 shows plots of the transmission time for one megabyte of data versus the x1, x2, x3, x4 and x5 variables (which are not independent). Both the observed and calculated values are shown in these figures. The calculated values show a good fit to the actual values. The same tests were performed using the Windows initiator and the UNH target. The results showed the same effect of pdu size on throughput, and the throughput values calculated using equation 2 were very close to the observed values.

As was done for reads, the same tests were repeated using the UNH initiator against vendor-A target, vendor-B target and vendor-C target. All the graphs showed the linear relationship given by equation 2, but with different coefficients calculated by a regression analysis in order to account for the different target back-end processing times.

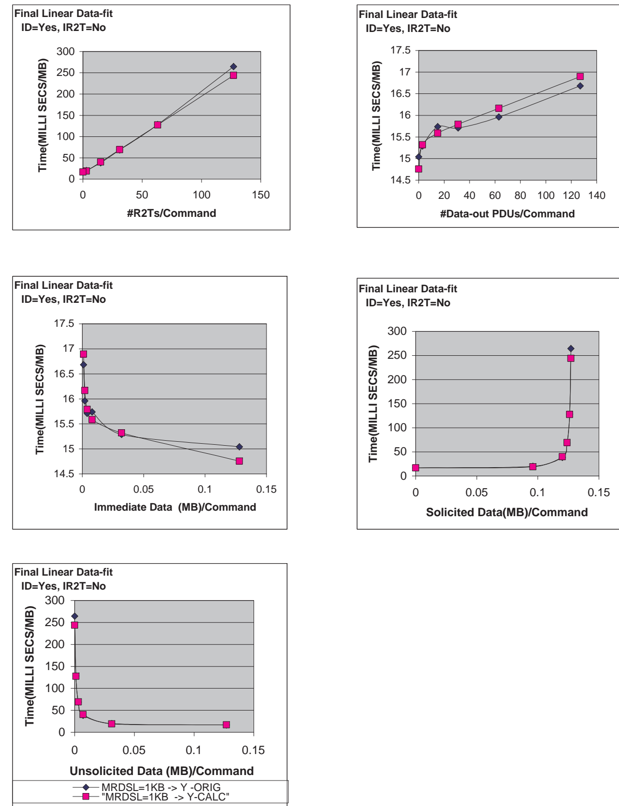

Figure 9. Observed values and calculated values for time/MB

The graphs in Figure 9 show the values with *ImmediateData* set to *Yes*, *InitialR2T* set to *No*, and *MRDSL* set to 1024 bytes. All other combinations also show a close match between the observed values and those calculated from equation 2.

6.3 Effect of multiple outstanding R2Ts

For this test *InitialR2T* was set to *Yes* and *ImmediateData* to *No* so that all data sent by the initiator was solicited by R2Ts received from the target. This test was repeated for various values of *MRDSL* between 512 bytes and 128KB. The burst lengths were kept equal to the *MRDSL* for each test in order to achieve the maximum number of R2Ts per transfer. The value of *MaxOutstandingR2T* was varied from 1 to 256.

Figure 10 shows that for any fixed value of *MRDSL*, as *MaxOutstandingR2T* increases the throughput increases up to a maximum limit and then levels off. The larger the number of outstanding R2Ts, the smaller the data transfer time because the initiator does not have to wait for another R2T to arrive at the end of each sequence. Hence, there is better bandwidth utilization. The graph also shows an interesting relationship: as *MRDSL* increases, the number of outstanding R2Ts required to get the highest throughput

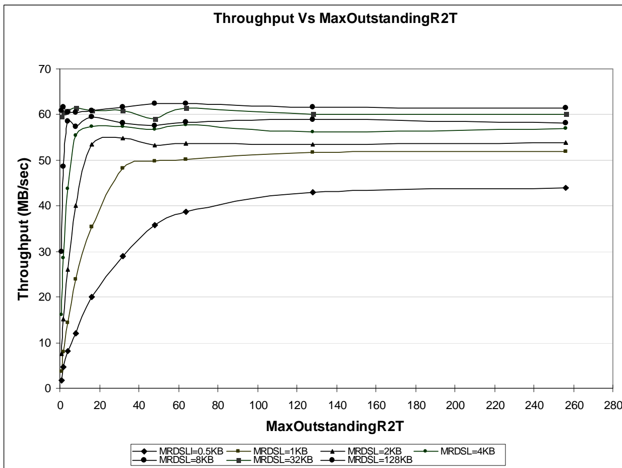

Figure 10. Effect of MaxOutstandingR2T on throughput

for that pdu size decreases. This can be approximated by the equation:

$$n*m = 64 \tag{3}$$

where:

n = Number of OutstandingR2Ts required to achieve maximum throughput for pdu size

m = MaxRecvDataSegmentLength (in KB)

	1	2	4	8	16	32	64	128	256
0.5	1.6704	4.584	8.1593	12.0708	20.1188	28.8984	38.6971	42.983	43.9762
1	3.6478	7.8801	14.3268	23.925	35.3707	48.3184	50.2325	51.8072	51.8749
2	7.655	15.2414	26.0099	40.0818	53.5416	54.7762	53.6337	53.4511	53.831
4	16.1368	28.6268	43.8112	55.484	57.3806	57.4192	57.6824	56.2435	56.975
8	29.9684	48.6475	58.4606	59.3547	59.5094	58.2166	58.3667	58.9518	58.1555
16	52.671	59.292	60.682	60.852	60.682	60.746	60.276	60.746	60.746
32	59.4938	60.5457	60.8546	61.4921	60.8346	60.924	61.4221	60.131	50.0875
64	60.345	61.321							
128	60.9504	61.7321	60.418	60.481	60.9442	61.6494	62.4659	61.6528	61.4988

Table 1. Data values for equation 3

Table 1 shows the test results with the values selected by equation 3 highlighted. This result might be specific to the UNH target implementation, but only that target could be used for this test because no other targets available to us currently implement more than one outstanding R2T. However, we expect this result is general because it illustrates that the R2T, which is intended to give the target control of its own buffers, also effectively controls the flow on the connection up to the point where the connection is running at maximum capacity. Outstanding R2Ts are analogous to buffer credits in other protocols, and this equation gives a method for determining the maximum amount of credit needed in iSCSI writes to achieve full utilization of the connection bandwidth.

6.4 Effect of multiple outstanding commands

Figure 11 shows the effect of increasing the number of outstanding SCSI write commands on throughput at various pdu and burst lengths. In this test *MRDSL* was varied from 512 bytes to 128KB, burst lengths were kept equal to the pdu size, and *MaxOutstandingR2T* was held constant at 1. For each pdu size the number of outstanding commands was varied from 1 to 64. The results show that for a fixed number of outstanding commands, larger pdu sizes produce higher throughput, and for fixed pdu size, more outstanding commands produce higher throughput. The graphs for small pdu sizes (512 and 1024 bytes) do not level off, but graphs for larger pdu sizes eventually reach a maximum beyond which increasing the number of outstanding commands does not produce any corresponding increase in throughput. However, the relationship is not as simple as that for *MaxOutstandingR2T* given in equation 3.

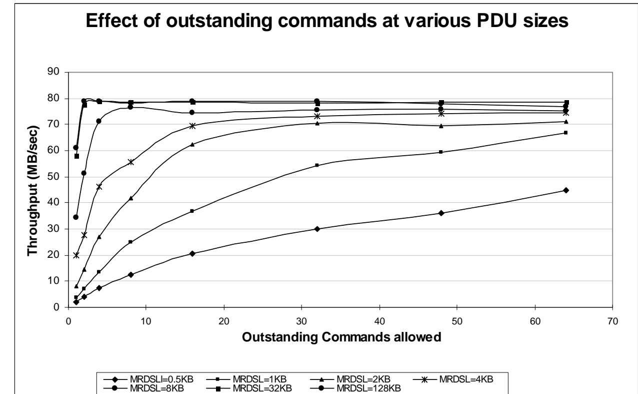

Figure 11. Effect of multiple outstanding commands

7 Conclusions and future work

Table 2 summarizes the conclusions of this study, and shows the suggested parameter settings that produce the best performance based on our observations. Note that in several cases the best values are not the default values specified in the iSCSI standard [2]. In general, one can conclude that small pdu and burst sizes should be avoided, immediate data and unsolicited data-out pdus are beneficial, and the number of sectors per command, number of outstanding commands and number of outstanding R2Ts should be maximized.

Further testing needs to be done on the items marked in Table 2, and to determine additional co-dependencies between parameters. If possible, the linear equations given in this paper should be extended to include additional factors such as target processing time. Initiator scheduling policies to dispatch commands across multiple connections in

Parameter	Best value	Default	Importance reads	Importance writes
MaxRecvDataSegment Length	>= 8KB	8KB	Important	Important
MaxBurstLength	>= 32KB	256KB	Not important	Important
FirstBurstLength	>= 32KB	64KB	Not relevant	Important
ImmediateData	Yes	Yes	Not relevant	Important
InitialR2T	No	Yes	Not relevant	Important
MaxOutstandingR2T	>= (64/MRDSL)	1	Not relevant	Important
Outstanding commands	>= 64	-	Important	Important
MaxConnections	Does not matter	1	Needs further testing	Needs further testing
Num sectors/command	1024	-	Important	Important
Digest	Off	Off	Important	Important
Phase collapse	Does not matter	-	Not Important	Not relevant
Number of LUNs on one connection	Does not matter	-	Not important	Not important
Scheduling policy (RR, LUNA)	Does not matter	-	Not important	Not important
DataPDUInOrder	Does not matter	Yes	Further testing required	Further testing required
DataSequenceInOrder	Does not matter	Yes	Further testing required	Further testing required

Table 2. Suggested iSCSI parameter settings

response to dynamic traffic conditions need to be implemented and studied. Studies with different types of storage devices are essential, as are tests using newer platforms with 64-bit buses and multi-processor configurations. Finally, a more detailed study of the interaction between iSCSI and TCP needs to be undertaken, with a goal of enabling iSCSI to dynamically set TCP parameters to improve performance.

References

[1] T10 Technical Committee of the NCITS. SAM2, SCSI architecture model – 2. Technical Report T10 Project 1157-D Revision 23, National Committee for Information Technology Standards, March 2002.

[2] J. Satran et al. Internet small computer systems interface (iSCSI). Technical Report RFC3720, Internet Engineering Task Force (IETF), April 2004.

[3] Kalman Z. Meth and Julian Satran. Design of the iSCSI protocol. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 116–122, April 2003.

[4] Stephen Aiken et al. A performance analysis of the iSCSI protocol. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 123–134, April 2003.

[5] Kalman Z. Meth. iSCSI initiator design and implementation experience. In *Proceedings of the 10th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 297–303, April 2002.

[6] Peter Radkov et al. An experimental comparison of block- and file-access protocols for ip-networked storage. In *Proceedings of the Usenix Conference on File and Storage Technologies*, March 2004.

[7] Ismail Dalgic et al. Comparative performance evaluation of iSCSI protocol over metro, local, and wide area networks. In *Proceedings of the 21th IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2004.

[8] University of New Hampshire InterOperability Laboratory. *iSCSI Initiator and Target Reference Implementations for Linux*. http://sourceforge.net/projects/unh-iscsi.

[9] Microsoft. *iSCSI Software Initiator Version 1.04*. http:// www.microsoft.com/ downloads/ details.aspx?FamilyID= 12cb3c1a-15d6-4585-b38-befd1319f825 &displaylang=en.